2

IDA PAPER P-2143

STRATEGY FOR ACHIEVING Ada-BASED HIGH ASSURANCE SYSTEMS

Richard A. DeMillo
R. J. Martin
Reginald N. Meeson

December 1988

Prepared for
Ada Joint Program Office
Rome Air Development Center

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 88-033705

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1988	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE
Strategy for Achieving Ada-Based High Assurance Systems

5. FUNDING NUMBERS
MDA 903 84 C 0031
T-D5-304

6. AUTHOR(S)
Richard A. DeMillo, R.J. Martin, Reginald N. Meeson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Defense Analyses (IDA)
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2143

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office (AJPO)
Room 3E114, The Pentagon
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

IDA Paper P-2143, Strategy for Achieving Ada-Based High Assurance Systems, documents the results of an analysis of software testing and verification technology conducted for the Ada Joint Program Office (AJPO) and the Rome Air Development Center (RADC) by the Institute for Defense Analyses (IDA). The paper presents a coordinated strategy for meeting a critical technology goal of the U.S. Department of Defense---the development of computer software for those systems upon which the Armed Forces can rely for the success of missions with extreme and often life critical requirements.

14. SUBJECT TERMS
Ada Programming Language; High Assurance Systems; Technology Insertion; Standards; Software Testing and Verification; Fault Tolerance.

15. NUMBER OF PAGES
82

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

IDA PAPER P-2143

STRATEGY FOR ACHIEVING Ada-BASED HIGH ASSURANCE SYSTEMS

Richard A. DeMillo
R. J. Martin
Reginald N. Meeson

December 1988

IDA
INSTITUTE FOR DEFENSE ANALYSES
Contract MDA 903 84 C 0031
Task T-D5-304

EXECUTIVE SUMMARY

Purpose

More than any other nation, the US has entrusted its military future to technology. Essential components of virtually every modern military system are the computers that support command and control, aim weapons, track and identify hostile targets, keep combat aircraft in stable flight, and automate many hundreds of other tasks. Essential to these computers is the software that controls their operation. This paper presents a coordinated strategy for meeting a critical technology need of the US Department of Defense (DoD)—the development of computer software upon which military missions can fully depend.

Recognizing the key role that computer software plays in modern weapon systems, the DoD and Congress have launched initiatives aimed at increasing national capabilities in software technology. The centerpiece of these initiatives is the programming language Ada. These initiatives are succeeding. Ada is now the cornerstone of software technology in DoD. However, the very success of Ada has highlighted the risk that software presents in military systems.

Need for Coordinated Strategy

In applications such as avionics, security, and strategic defense, it is not enough to design the required capabilities into the system—systems must be *assured* of meeting their requirements. Systems must undergo such indisputable engineering analysis that the risk of failure is virtually eliminated. In the modern military world, high levels of assurance must be demonstrated before rational decision-makers can field and operate advanced automated systems. Systems that combine extreme operational and engineering requirements with severe assurance requirements are called *high assurance systems*.

In spite of the importance of high assurance systems within the DoD, assurance technology development and management is largely fragmented and ineffective. At senior levels within the DoD, software is still viewed as an uncontrolled risk element. Important assurance programs are scattered and uncoordinated. Existing technology is rarely shared among programs or Services. Research and development expenditures for software assurance technology has declined and the research community has dwindled.

Policy and guidance advances of the early 1980s have not been pursued vigorously. Contracts rarely include explicit and unambiguous assurance requirements. In some instances, assurance is not attended to at all until very late in the acquisition process. Regulations and standards give inadequate guidance to developers of high assurance systems. New software standards and regulations have actually weakened assurance requirements, not strengthened them. Standard terminology is nonexistent, giving free reign to contractors to manipulate system evaluations.

While Ada has been established as the basis for mission critical software in programs ranging from the Strategic Defense Initiative to the Advanced Tactical Fighter, the assurance technology needed to support Ada has been neglected. If this lack of attention is not reversed, Ada-based high assurance systems and, in turn, national security will be in jeopardy. The recommendations in this report should be implemented as essential parts of DoD research and engineering.

Recommendations

The following specific actions are recommended to achieve and sustain Ada-based high assurance systems:

Assurance Requirements —

- Review and, where necessary, revise existing acquisition policy and implementing materials to ensure that a controlled and manageable assurance process (called incremental assurance) is institutionalized.

Standards —

- Review and revise DoD-STD-2167A and DoD-STD-2168 to require the application of appropriate assurance methods to high assurance systems.
- Develop a standard software engineering glossary that is acceptable to all DoD components listing terms, definitions, and concepts that correspond to standard engineering usages.

Technology Insertion —

- Utilize existing technology insertion mechanisms for assurance technology.
- Incentivize contractors to develop and deliver productized versions of assurance tools.

Required Research —

- Establish a broadly-based research initiative to rebuild the assurance research community. This program should be coordinated with the National Science Foundation (NSF) and other Government funding agencies, and should have a goal of increasing funding at least four-fold over a five-year period.
- Direct DoD and Service Scientific Research Offices to fund both innovative assurance technology projects and longer-term projects aimed at maturing and validating existing methods.
- Develop joint industry-university research initiatives.

Coordination —

- Implement and institutionalize the technology and management recommendations in coordination with the Test and Evaluation community within the DoD.

PREFACE

The purpose of IDA Paper P-2143, *Strategy for Achieving Ada-Based High Assurance Systems*, is to communicate the results of an analysis of software testing and verification technology conducted for the Ada Joint Program Office (AJPO) and the Rome Air Development Center (RADC) under Task Order T-D5-263.

An earlier draft of this paper was reviewed within the Computer and Software Engineering Division (CSED) by T. Mayfield, K. Price, and R. Winner (August 31, 1988).

Acknowledgements

The authors are indebted to the following people for participating in planning meetings and helping to set the directions for this work, and for their comments on earlier drafts of this paper: W. Easton, J. Faust, P. Fonash, R. Gross, S. Hadley, G. Hoover, A. Kopp, K. Kucherary, T. Mayfield, K. Rowe, J. Salasin, D. Vaurio, and R. Winner.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 PURPOSE

More than any other nation, the US has entrusted its military future to Technology. A critical component of virtually every modern military system is the computer componentry that supports command and control, aims weapons, tracks and identifies hostile targets, keeps combat aircraft in stable flight, or automates many hundreds of other tasks. This report presents a coordinated strategy for meeting a critical technology goal of the US Department of Defense (DoD)—the development of computer software for these systems upon which the Armed Forces can rely for the success of missions with extreme (often life-critical) requirements.

1.2 SCOPE

The recommendations in this report cover four broad areas:

- Definitions and concepts: Section 2 presents terminology, definitions, and *basic engineering concepts. It is recommended that* DoD software technology programs adopt and promulgate these definitions to all DoD components.
- Available technology: Section 3 summarizes the state of the available technology, engineering methods, and tools that can be applied to provide software assurance. Gaps in current technology are pointed out and strategies are recommended for ensuring that mature assurance technology will be available for high assurance Ada-based systems.
- Assurance requirements: Section 4 discusses methods for identifying and specifying required levels of software assurance. These include identifying software risks, specifying required software characteristics, and assessing the effectiveness of assurance programs.
- Technology transition and coordination: Section 5 identifies efforts the DoD should initiate to achieve its mission-critical software assurance needs. This includes a high-level strategy for identifying the most promising assurance technologies and putting them to use in system development. Relevant recommendations are aimed at institutionalizing a coordinated effort whereby existing and future technology advances can be captured for use by DoD.

While these recommendations are directed primarily toward providing assurance for systems with critical safety and reliability requirements, the definitions and technology apply equally to less critical systems.

1.3 BACKGROUND

As Mission Critical Computer Resources (MCCR) become an increasingly common component of modern military systems, computer software continues to loom large as a major source of risk in system acquisition, operation, and maintenance. In 1984, an IDA study (Redwine et al. 1984), commissioned by the DoD, found that of the six existing long range planning documents that outlined future technologies for 1990 and beyond, 70 percent of the technologies, functions and systems described in those plans required software. A detailed analysis of required functional areas was carried out for one of these planning documents in order to identify the extent to which software is important to fulfilling mission objectives. This analysis showed that, above the guns and boots level of combat support, virtually all mission areas depended on software to fulfill 80 percent or more of their functions.

Furthermore, the software needed for these systems is often subjected to the most extreme requirements:

- In avionics applications, typical reliability and availability requirements specify software capable of delivering over *10^9 continuous service* hours without loss of critical system functionality.
- In security applications, multilevel secure systems capable of withstanding arbitrarily sophisticated penetration attempts comprise an important capability for the heavily networked command and control hierarchies of all the Services (DoD 1983).
- In strategic defense battle management/command, control, and communications (BM/C3) systems, software must discriminate, track, and reliably direct ordnance to hundreds of thousands of objects.

In these and many other applications, it is not enough to design the required capabilities into the system. These systems must not only satisfy their requirements, *they must be assured of doing so*. That is, the systems must have undergone such indisputable engineering analysis that the risk of failure has been eliminated. Or, if the risk of failure cannot be eliminated, then the likelihood of mission success must demonstrably balance the threat of failure. In each of the application areas mentioned above, a requisite level of assurance is needed before a rational decision-maker would consider operating the system. Systems that combine extreme operational or engineering requirements with severe assurance requirements are called *high assurance systems*.

In spite of the importance of high assurance systems in the DoD, assurance technology management is largely fragmented. Senior-level DoD assurance policy developers still view software as a significant and largely uncontrolled risk element. In development organizations, important assurance programs are scattered and uncoordinated. Standard terminology is essentially nonexistent, giving free reign to contractors to invent definitions and thus manipulate system evaluations. Existing technology is rarely shared among programs or Services. Regulations and standards give inadequate software guidance to developers of high assurance systems. The effect is that statements of work and contracts rarely include explicit and unambiguous assurance requirements—as a result most development efforts spend their resources on more glamorous engineering aspects, such as design. In some instances, assurance is not attended to at all until very late in the acquisition process.

These observations were initially documented in 1983; for further elaboration, see (DeMillo et al. 1987). In the intervening years, the situation has probably deteriorated. Research and Development (R&D) expenditures for software assurance technology have declined. The research community has dwindled. New software standards and regulations weaken assurance requirements. Even the policy advances of the early 1980's (Bolino and McCracken 1984) have not been pursued with vigor.

The advent of the Ada language has had many benefits for software engineering and will undoubtedly ease many of the productivity and quality concerns that have given rise to the DoD software initiatives. However, the Ada community has concentrated its efforts on compilers, environments, and development tools. Despite a few well-publicized attempts to develop some momentum (Roby 1985), assurance technology for Ada has definitely taken a back seat. Given the general state of disrepair of software assurance technology and the lack of attention given to assurance technology in the Ada community, there is a chance that high assurance Ada-based development for the early 1990's will be in jeopardy.

1.4 LIST OF ACRONYMS

DoD	Department of Defense
IDA	Institute for Defense Analyses
MTBF	Mean Time Between Failures
NCSC	National Computer Security Center
OSD	Office of the Secretary of Defense
R&D	Research and Development
T&E	Test and Evaluation

2. TERMINOLOGY, DEFINITIONS, AND BASIC ENGINEERING CONCEPTS

2.1 DEFINING ASSURANCE

The literal meaning of the verb *to assure* is *to make safe*. The term also connotes evidence of certainty in the form of a legal certificate or insurance policy. Even in everyday usage, *assurance* is a complex concept. In this paper, the safety concept is used extensively to summarize assurance goals. The safety metaphor is general, as demonstrated below. To convey assurance requires:

- A specification of the *hazard* or *threat*. Safety is a relative notion. It is not possible to be absolutely safe. Rather, safety must be judged with respect to a specified hazard or threat to safety.
- A *certificate* of certainty that functions like an insurance policy (or other legal assurance). In other words, the certifying agent must have made an assessment of the risk in terms of the likelihood of the threat, the probable loss if the threat succeeds, and the likelihood that the threat will succeed. Then, taking into account the expected error in the risk assessment process, the certifying agent estimates the extent to which he is willing to certify that he is certain of safety.

For mission critical systems, assurance requires:

- Specification of *properties*: including any characteristics of the system to be demonstrated or maintained when faced with given hazards or threats. Examples of such properties will be given later.
- Identification of threats: explicit specification of the conditions under which the property is to be maintained. In some cases, the "threat" corresponds to the hazards that are encountered in the system's intended environment. Other times, it may be useful to deal with hazards that are due to engineering shortcomings leading to system vulnerabilities.
- Certification of assurance: an explicit expression of confidence that the system will be maintained in the given property when confronted with the threat or hazard. Like an insurance policy, this certificate can have high cost if the risk is high (either the vulnerability is great, the hazard is significant or the cost of loss is large).

2.2 PROPERTIES, HAZARDS, AND VULNERABILITIES

It is usual practice in system engineering to combine the property with the hazard and call the composite concept the "property." This is to emphasize that the properties are not absolute, but rather are relative to some specified hazard. For example, the following is the usual definition of the reliability property:

> Reliability is the probability that a system will perform as intended under stated conditions for a specified period of time.

The following are important aspects of such property definitions:

- *The threat or hazard is explicit*: the definitions require the system to be operated in a specified environment by personnel typical of those who will use the system in practice and for a specified length of time. Any failure to perform intended functions within this envelope is a reliability event that reduces the measured reliability of the system. Notice that no requirements whatsoever are placed on the system when it operates outside this envelope (although sometimes failure types include "Acts of God" or other ways of making the envelope very large).
- *The definition implies measurement criteria*: in fact, the definition serves as a guideline for how to conduct tests and measurements to determine whether or not the property is satisfied. Sometimes definitions that include their own measurement criteria are called *testable*. By referring to the specified functions, the definition requires a list or other characterization of the acceptable operating capabilities. Furthermore, the definition explicitly requires a probability distribution on the failures (or non-failures). The definition is explicit in that to satisfy the property, the system when operated as specified must exhibit a mean time between failures (MTBF), calculated by dividing the observed reliability events into the specified mission time. The failure distribution is then modeled as a rate of failure per unit time, 1/MTBF, sometimes called the hazard rate (the reliability is then 1-1/MTBF).

By the same token, systems may have inherent weaknesses that can be exploited by hazards. The usual term for this is *vulnerability*. That is, a system vulnerability is a system property that increases likelihood of hazard success.

2.3 OPERATIONAL AND TECHNICAL PROPERTIES

Reliability is an example of an operational property. That is, its hazard is defined with respect to the operational threat and associated environment. Other properties take on meaning without reference to the operational environment. Consider the following definition:

> Correctness is the extent to which the software satisfies its specified requirements, or the extent to which the software conforms to its specifications and standards.

Like the definition of reliability, this definition is testable. It implies a method of measurement by requiring that the *extent* of conformance to a fixed set of specifications be measured. Unlike reliability, however, the hazards of incorrect software do not arise from the operational environment. They are technical problems that arise from the engineering process. The threat here is that faults in the software may prevent the system from conforming to its specifications.

Technical requirements and their associated properties refer to various engineering parameters, specifications, tolerances, and other aspects of the engineering process that may influence overall product effectiveness. Correctness is a technical property since it attempts to place a requirement on the output of the engineering process. The hazard consists of whatever defects exist in the engineering process. Measurement methods for technical requirements include checklists, traceability charts, test coverage, or any similar devices that allow one to track extent.

Since the technical hazard exists independent of any particular external threat (e.g., an error or "bug" exists in the software, whether or not it is ever stimulated by an input), it represents an inherent weakness in the system. Therefore, it is common to use system vulnerability as the measure of technical hazard.

2.4 SETTING REQUIREMENTS ON PROPERTIES

Technical and operational properties must have:

- Explicit identification of the hazard
- Testable definitions

The properties are used to set technical and operational requirements. These requirements are constraints that the system must satisfy in order to conform to necessary engineering standards, tolerances, or quality criteria; to be operationally functional or effective against an external threat; or to operate with suitable non-functional characteristics in its intended environment.

Requirements can be stated in either of two forms:

- *Goals*: desired or expected values of technical or operational properties and parameters. Goal requirements are also used in evolutionary developments to indicate a requirement toward which the system will grow.

- *Thresholds*: values of technical or operational properties and parameters below which overall system worth will be unacceptable.

For purposes of applying assurance technology, threshold requirements are more valuable than goals. With a threshold requirement, the risk can be directly calculated from the probability of not meeting mission objectives as specified in the relevant properties. If a goal requirement is satisfied, then clearly the risk is reduced. On the other hand, if a goal requirement is not satisfied, the assurance agent has a difficult time estimating overall system worth—the goal does not set minimum engineering or operational criteria.

2.5 EXAMPLES OF PROPERTIES

The following properties are commonly used in requirements statements for mission critical Ada-based systems. Many of these definitions have been taken directly from existing DoD documents. However, in those cases where the existing definition is defective (e.g., does not identify an explicit hazard or is not testable) the definition has been repaired without altering its intent. In those cases where there are conflicting definitions, the usage that is most consistent with standard engineering usage has been chosen. Notice that, in some circumstances (e.g., security) the property is used ambiguously to designate the extent to which measures have been taken to make the system safe as well as the likelihood of success of a specified threat. This is important since assurance of the technical property does not necessarily imply assurance of the operational property.

Software quality is the degree to which software possesses a desired combination of attributes (i.e., technical and operational properties). Hence, it is not identified as a separate property.

2.5.1 Technical Properties

The following technical properties are used in requirements statements for mission critical Ada-based systems:

- *Cohesion*: the extent to which specified data names, structures, and functions in a software component depend upon each other.
- *Correctness*: the extent to which the software satisfies its specified requirements, or the extent to which the software conforms to its specifications and standards.
- *Coupling*: the extent to which data names, structures, and functions in different software components depend on each other.
- *Security*: the extent to which specified measures have been taken to protect computer hardware, software, and resident information and data from

unauthorized access, use, modification, destruction, transmission, or disclosure.

2.5.2 Operational Properties

The following operational properties are used in requirements statements for mission critical Ada-based systems:

- *Availability*: the probability that a specified function or capability can be initiated or invoked when the system is operated in its intended environment for a specified period of time.
- *Fault-Tolerance*: the probability that a system detects, recovers and insulates itself from the effects of specified component faults or failures in order to maintain a high degree of availability when operated under stated conditions for a specified period of time.
- *Integrity*: the probability that stored information and data will not be modified by specified unauthorized means.
- *Maintainability*: the probability that specified unavailable functions can be repaired or restored to their operational state in the system's intended maintenance environment during a specified period of time.
- *Reliability*: the probability that the software will perform as intended under stated conditions for a specified period of time.
- *Security*: the probability that computer hardware, software, and resident information and data will be protected from specified threats such as unauthorized access, use, modification, destruction, transmission, or disclosure.
- *Survivability*: the probability that the software will perform and support critical functions in its intended environment without failure when a specified portion of the system is inoperable.

2.6 ASSURANCE METHODS

The overall goal of an assurance method is to reduce the risk associated with a software system and one or more properties. The risk is determined by the likelihood of success for the hazard, the level of confidence that the assurance method leads in fact to a specified reduction in probability of success for the hazard, and the relative costs of (1) the assurance method and (2) of failure to satisfy the property.

There are three ways that an assurance method can reduce the probability that the hazard will succeed:

- Eliminate the hazard.
- Decrease the likelihood of success for the hazard.

- Demonstrate that the relevant property can be maintained even in the presence of the hazard.

The effectiveness of the assurance method is determined by two factors:

- Coverage of the hazard: it is possible for an assurance method to address all occurrences of a hazard, only a portion of the hazard, or no occurrences of the hazard.
- Level of confidence in the method: even if the assurance method produces a positive result, it is still possible that the software will ultimately fail to satisfy a given property. The probability of this occurrence is measured by the level of confidence in the method.

2.6.1 Eliminating Hazards

This approach to assurance seeks to decrease the likelihood of a hazard ever occurring. One common assurance method that works this way is the application of systematic design methods. The hazard in this case is the occurrence of one or more software design, specification, or implementation errors. Another technique is to design the system to be adaptable to changing threats. The hazard in this instance is the occurrence of a threat that may not be fully anticipated or understood by the designers.

2.6.2 Decreasing the Likelihood of Hazard Success

This is the traditional domain of testing, verification, and validation. Systematic assurance methods can identify specific system vulnerabilities and estimate the probability that a given hazard will be able to successfully exploit the vulnerability. An operational reliability test, for example, subjects the system to realistic scenarios of the kind that will be encountered in actual usage, records and classifies reliability events, and calculates the expected system reliability. If the system vulnerabilities make it unsuitable for its mission, then a redesign or maintenance action is undertaken to remove or reduce the vulnerabilities.

2.6.3 Tolerating Hazards

The prime example of this approach is fault-tolerance to assure system availability. The hazard in this case is the occurrence of one or more faults or failures in system components. The design must be capable of detecting the occurrence of a fault, containing the effects of the fault, and invoking the required function in a manner that avoids the fault. For fundamental reasons, this approach requires some sort of redundancy and therefore may increase overall system cost.

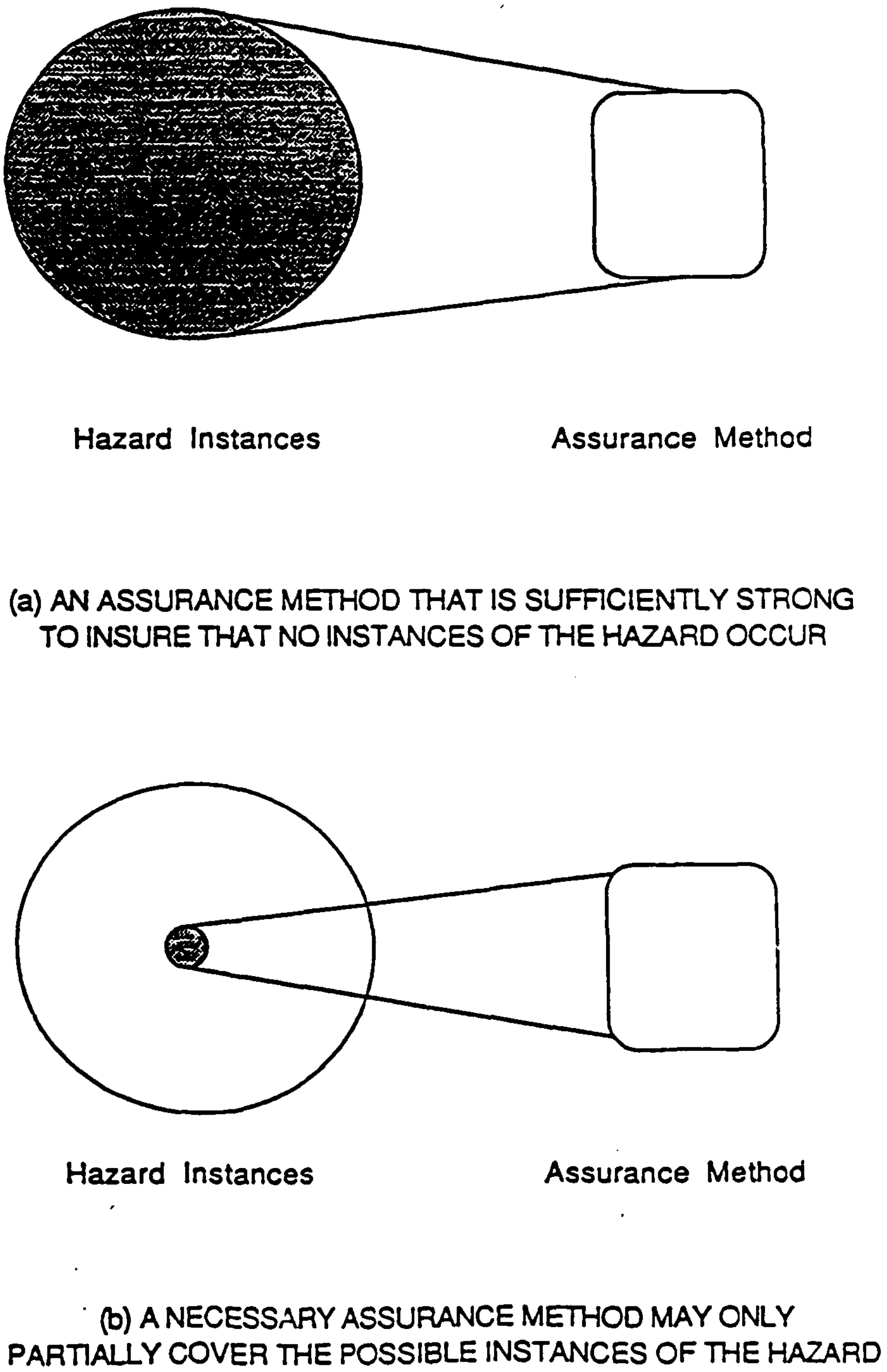
2.6.4 Coverage of Hazards

A key aspect of every assurance method is the extent to which it covers the hazards. Some methods offer complete coverage. These methods are sufficient to guarantee the property of interest. Suppose, for example, that the property to be assured is correctness with respect to a fixed specification of security. If a valid proof of correctness exists for that specification, then no software error can exist that will produce an inconsistency with the specification. The space of all such hazards has been completely covered by the proof of correctness. Since assurance methods that completely cover the space of hazards are sufficient to guarantee the property, they will be called *sufficient assurance methods* (see Figure 1a).

Another alternative is that the method only partially covers hazards (Figure 1b). For many properties of interest in Ada-based mission critical applications, these methods play a crucial role. In the example above, the proof of correctness is sufficient but may not be immediately useful (e.g., the proof may not be readily obtainable or the level of confidence in the validity of the proof may be unacceptably low). In these circumstances, it is often desirable to turn to necessary methods, even though they only partially cover the hazards. In this case, a test that executes every statement in the software is such a method. Passing the test covers only those errors that can be revealed by complete statement coverage (a small subset of all possible errors). Not conducting such necessary tests, however, significantly raises the risk of errors in the unexecuted portions of the software. In fact, without actually executing all statements it is generally not possible to rule out the presence of viruses, covert channels, and other threats and vulnerabilities. Complete statement coverage is, in other words, a *necessary* assurance method (even though it is usually not sufficient).

2.6.5 Levels of Confidence

If an assurance method provides a fixed indication that the software satisfies its intended property, there is somuimcs a residue of doubt about whether or not the property actually is satisfied. Sometimes tnis residue is so small that it is negligible. In certain applications, for example, there are only a relatively small, finite number of ways to stimulate the software, so that an exhaustive test of such a system and the proper instrumentation to measure whether or not the property holds gives an extremely high *level of confidence* in the result. By contrast, it is usually not possible to conduct such an exhaustive test. One alternative is to generate random sequences of sample inputs according to probability distributions that are known or can be assumed to exist in the operational environment. In these cases, there is a distinct likelihood that (a) the chosen distribution for the input space is incorrect, or (b) the sample size was too small, or (c) the



(a) AN ASSURANCE METHOD THAT IS SUFFICIENTLY STRONG TO INSURE THAT NO INSTANCES OF THE HAZARD OCCUR

(b) A NECESSARY ASSURANCE METHOD MAY ONLY PARTIALLY COVER THE POSSIBLE INSTANCES OF THE HAZARD

Figure 1. Coverage of Hazards by Assurance Methods

operational scenario rules out the use of random sampling of inputs. In security applications, for example, the most threatening opera:ional scenarios are far from random—they are chosen by purposeful adversaries who exploit known or suspected vulnerabilities. These factors all work to lower the level of confidence in the assurance method. High levels of assurance may result from single applications of assurance methods with high levels of confidence or from the combined application of such methods.

One way to select an assurance method is by its *utility*. Using this strategy, one would choose to apply the method that yields the highest levels of confidence. Another strategy is to select the method according to a *risk-aversion* rule. Using this rule, one would select methods that would provide sufficient confidence for hazards or threats with the highest risk. In practice, combinations of these selection rules are used based on the various other risk factors described above.

2.6.6 Classifying Assurance Methods

The next section of this report will briefly review the major assurance methods that are applicable to Ada-based systems. These methods will be classified according to how they address the hazards, the levels of assurance that are possible and the kinds of assurance properties they address. Table 1 summarizes this classification.

Table 1. Assurance Method Classification

Legend:
S = Sufficient Method
N = Necessary Method
L = Low Certainty
H = High Certainty
X = Method Applies to Indicated Property

		METHODS			POWER		TECHNICAL PROPERTIES				OPERATIONAL PROPERTIES						
		ELIMINATING HAZARD	DETECTING/REMOVING HAZARD	TOLERATING HAZARD	HAZARD COVERAGE	LEVEL OF CONFIDENCE	CERTIFIABILITY	CORRECTNESS	QUALITY	SECURITY-TECHNICAL	AVAILABILITY	FAULT-TOLERANCE	INTEGRITY	MAINTAINABILITY	RELIABILITY	SECURITY-OPERATIONAL	SURVIVABILITY
SPECIFICATION	PROTOTYPING	X	X		S	L					X	X	X	X	X	X	X
	EXECUTABLE SPECIFICATIONS	X	X		S	L					X	X	X	X	X	X	X
	MULTIPLE SPECIFICATIONS	X	X		?	L		X									
	FINITE STATE METHODS	X			S	H		X									
	PETRI NETS	X			S	H		X									
	FORMAL SPECIFICATIONS	X	X		S	H		X		X				X			
	ADA PDL	X	X		S	H		X		X				X			
	REQUIREMENTS MODELLING	X	X		S	L					X	X	X	X	X	X	X
DESIGN/PROGRAMMING	STEPWISE REFINEMENT	X	X		S	L		X	X					X			
	STRUCTURED DESIGN	X	X		S	L		X	X					X			
	OBJECT ORIENTED DESIGN	X	X		S	L		X	X					X			
	STRUCTURED PROGRAMMING	X	X		S	L		X	X					X			
	OBJECT-ORIENTED PROGRAMMING	X			S	L		X	X					X			
	AUTOMATIC PROGRAMMING	X			S	H		X									
TESTING	STRUCTURED WALKTHROUGHS		X		N	L		X	X					X			
	SYMBOLIC EXECUTION		X		S	H		X		X							
	STRUCTURAL COVERAGE		X		N	L	X	X		X							
	DOMAIN COVERAGE		X		N	L	X	X		X							
	MUTATION TESTING		X		S	H		X		X					X		
	FUNCTIONAL TESTING		X		N	L	X	X		X							
	RANDOM TESTING		X		S	H		X		X					X		
	OPERATIONAL TESTING		X	X	N	H	X		X		X	X	X	X	X	X	X
FORMAL VERIFICATION	GYPSY		X		S	H	X	X		X							
	VDM		X		S	L	X	X		X							
	IV&V		X		S	L	X	X	X	X				X			
POST DEPLOYMNET	PROBLEM TRACKING		X	X	N	L		X	X	X	X	X	X	X	X	X	X
	OBSERVATION PACKAGES		X	X	N	L		X	X	X	X	X	X	X	X	X	X
	METRICS	X	X	X	?	L			X								
FAULT TOLERATNCE	RECOVERY BLOCKS			X	S	H					X	X			X		
	N-VERSION PROGRAMMING			X	S	H					X	X			X		
OTHER METHODS	EVOLUTIONARY DEVELOPMENT	X	X	X	N	L		X	X	X	X	X	X	X	X	X	X
	REUSE	X	X	X	N	L		X	X	X	X	X	X	X	X	X	X

3. AVAILABLE TECHNOLOGY

The assurance technology available for technical properties is almost exclusively error and fault based. In other words, this technology explicitly addresses the correctness property. Other properties must be restated in terms of correctness before these methods apply.

3.1 FAULT AVOIDANCE

The first step in the development of Ada-based high assurance software is to avoid the initial introduction of faults into the software. This section will review software engineering technology that addresses the specification, design, and construction of high assurance software.

3.1.1 Specification Technology

A prerequisite to the development of high assurance software is a precise understanding of its intended performance. In other words, software should not be built to satisfy unknown requirements. This applies to assurance requirements, as well as to technical and operational performance requirements. If the software is to achieve certain levels of assurance, the associated requirements must be specified prior to the software's design and construction. Operational and technical properties of interest were defined earlier. The problem is how to adequately specify these quality requirements.

One way to determine requirements is to build prototypes of key portions of the system for examination by application specialists. Prototypes facilitate communication between system users and system developers (Henderson 1986). The software development process consists primarily of the software developer documenting and constructing his interpretation of the users' system concept; however, seldom do the backgrounds and terminologies of the system user and the software developer coincide. Therefore, tools and techniques that aid in the unambiguous transfer of information and requirements from user to developer are of prime importance. In addition, prototypes can demonstrate the feasibility of system requirements and provide a basis for experimentation during trade-off analyses (Aho, Kernighan, and Weinberger 1985). Another vehicle for user-developer communication is application-oriented specifications (Bentley 1986).

Executable specifications and the creation of multiple specifications are other techniques for uncovering errors in the initial documentation of system requirements. Executable specifications allow testing and validation of the requirements prior to detailed design and implementation (Zave and Schell 1986). When inconsistencies exist, the users are queried for the information necessary to settle the dispute. When variations represent alternate conceptualizations of the system, they can form the basis for future trade-off studies or redundant development efforts.

Each of the techniques described above concentrates on the examination of the system requirements as represented by the specifications. In one sense, this is a look backward in the system development process. Specification technology also encompasses a variety of computing oriented specification techniques that emphasize the forward look into the software development process. Formal computing oriented specification notations can be categorized as either state-oriented or relational notations (Fairley 1985). State-oriented notations include decision tables, finite state machines, and Petri nets. Petri nets are of special interest since they support the specification of concurrency (Peterson 1977; Bruno and Marchetto 1986). State charts combine the best of several specification methods and are particularly useful for specifying real-time systems (Bruns et al. 1986). Relational notations include formal languages (both graphical and textual) which allow automatic analysis for completeness, consistency, and satisfaction of other specification rules. Examples of specification languages and processors include PSL/PSA (Teichrow 1977), RSL/SREM (Alford 1977), and Gist (Balzer 1981). The use of Ada for specification purposes has also been investigated (Linn et al. 1988).

In contrast to the formal specification techniques described above, fuzzy specification techniques can be used to avoid the over-specification of requirements. This technique incorporates imprecision in situations where additional accuracy does not contribute to the ultimate solution of the problem (Rine 80). This technique could be quite promising as a method for the effective utilization of the scarce resources allocated for requirements specification.

The anticipated life times of today's Ada-based systems and the certain changes in their operational environments make software engineering technologies that support evolution very important. The evolutionary process pivots on the continuing evaluation and improvement of the specifications over time. Essential to the control of this process is a mechanism for retaining information about the specifications for past, present, and future instantiations of the software (Martin 1987).

Other aspects of contemporary Ada-based systems that demand advanced specification technology include the requirements that they be real-time, distributed, fault-tolerant, and secure. Real-time and distributed system requirements necessitate the ability to represent timing, synchronization, and ordering information in the specification. One example of work in this area is Communicating Sequential Processes (CSP) (Hoare 1978). Fault-tolerant and security requirements necessitate the ability of the specifications to represent information about the fault model being employed, as well as security model properties (DeMillo and Merritt 1983). Finally, Ada compounds the need for a specification technique that supports concurrency and tasking.

3.1.2 Design Technology

Once the requirements have been determined and appropriately specified, software engineering design technology exerts its influence on the ultimate quality of the application. The fundamental concepts of software design include abstraction, information hiding, structure, modularity, and concurrency. An assortment of design techniques and notations are currently available. They can be distinguished by the levels of emphasis placed on each of the fundamental concepts. Examples of design techniques include stepwise refinement (Wirth 1971), structured design (Yourdon and Constantine 1979), object-oriented design (Booch 1983), and Jackson System Development (Jackson 1982). These techniques employ design notations such as data flow diagrams, structure charts, procedure templates, and design languages.

Selections of design technology for application to specific projects are made subjectively due to the lack of objective evidence differentiating the effectiveness of the techniques. In fact, it appears that the major benefit derived from the use of modern design techniques stems from the enforcement of a systematic approach rather than inherent characteristics of the techniques. Further, experimental results challenge the benefits of adhering strictly to particular design concepts such as modularity and information hiding (Card, Church, and Agresti 1986).

As with specification technology outlined previously, the selection of technology for use during the design of software must be guided by the requirements of the application. Again, any techniques selected must be able to represent timing, synchronization, and ordering information. Appropriate design techniques for use with Ada, such as data abstraction and object-oriented design, are also important (Booch 1983). Finally, the evolution of software requires that chosen design techniques facilitate the process of change.

3.1.3 Programming Technology

Current programming technology, in many cases, parallels current design technology. Modern programming language features support the fundamental concepts of design: data abstraction and separate compilation support design abstraction, scoping rules support information hiding, coding standards support structured design and modularity, and concurrency supports parallelism in designs. Further, many of the design techniques reviewed previously are implemented by currently available programming techniques: structured programming (Linger, Mills, and Witt 1979), object-oriented programming (Shriver and Wegner 1987), and Jackson Structured Programming (Jackson 1982). And, as with design, it is not apparent that any of the currently advocated programming techniques is clearly superior to the others with respect to the ultimate quality of the application software.

The application of artificial intelligence to the realm of software development has resulted in a variety of attempts to either automate a portion of the programming process or assist human beings with the tasks associated with programming. The term applied to this area of research is *automatic programming*. Programming techniques that reduce the dependence on humans are attractive since human error is thought to be a major cause of low quality in software systems. Automatic programming systems currently under investigation can be distinguished by their specification methods and approaches (Barr and Feigenbaum 1982). Specification methods include the use of formal specification, specification by example, and natural language specification. Approaches to program generation include theorem proving, program transformation, knowledge engineering, and traditional problem solving. Examples of experimental systems include PSI (Green 1976), SAFE (Balzer, Goldman, and Wile 1976), the Programmer's Apprentice (Rich and Shrobe 1978), and Protosystem I (Ruth 1978).

3.2 FAULT LOCATION AND REMOVAL

As software products become available, the emphasis of the software development process shifts from fault avoidance to locating and removing the faults that were introduced in spite of the use of fault avoidance techniques. Three primary types of activities undertaken are test and evaluation (T&E), quality assurance (QA), and verification and validation (V&V). Testing techniques are usually classified as either static or dynamic analysis (DeMillo et al. 1987). When utilizing static analysis techniques, the software is usually not executed; rather, human code readers or automated analyzers process program text in order to resolve outstanding issues. Dynamic analysis techniques, on the other hand, depend on code execution for data concerning the software's behavior.

3.2.1 Static Analysis

Static test methods are primarily used during early design stages to verify the consistency of intermediate engineering or incomplete software products with prior specifications or other documents. The nature of static methods makes them ill-suited for directly addressing operational properties or issues. Nevertheless, static analysis is the principle tool for deriving early estimates of whether or not required technical properties are satisfied. Some of these methods simply measure the extent to which basic engineering standards have been satisfied. Still others carry out more sophisticated analyses (e.g., identifying statements that cannot be reached by any feasible control path).

An important class of static methods involves the structured "reading" of program instructions. These readings or walk-throughs can be carried out in a group format (during which the group may play an adversarial role) or by a single programmer. Three major issues that can be resolved with this technique are the following:

- Completeness: is every specified requirement successfully addressed in the design?
- Consistency: is the design consistent with itself as well as with previous specifications and constraints?
- Traceability: are all elements of the design traceable to specific required functional features or capabilities?

One difficulty in relying on static analysis, particularly those approaches utilizing subjective evaluations or human readers, is their relative non-reproducibility. Although such tests can be planned and carried out to support specific test objectives, test reporting and sharing of test results may not be possible.

Static analysis techniques as applied to both design documents and code have benefited from extensive examination and application in recent years. Although tools implementing static analysis techniques are not yet abundant for Ada, the underlying technology is mature and it is doubtful that automation of desired capabilities would pose many problems.

Symbolic execution lies in the gray area between static analysis and dynamic analysis. This technique assigns symbolic values to program variables and produces an execution tree which characterizes the possible execution paths. If each path can be shown to be symbolically correct (i.e., the symbolic expression associated with the execution of the path corresponds to the specifications), then the program is correct (Darringer and King 1978). The primary uses of symbolic execution are test data generation and proving program correctness.

3.2.2 Dynamic Analysis

The application of dynamic analysis techniques is essential for Ada-based high assurance systems. Determination of the software's satisfaction of extreme quality requirements can only be accomplished by systematically exercising the software.

Principal dynamic analysis activities are the design of tests to establish certain characteristics of the software, the execution of the tests according to plan, and the analysis of results as allowed by the underlying theory of the chosen testing technique. Testing techniques of interest include structural coverage tests, domain testing or input space partitioning, error coverage tests, functional testing, random testing, and macroscopic program instrumentation.

A structural coverage test is satisfied if test data can be supplied that causes the execution of the specified percentage of the structural features of the software. A common structural coverage measure is statement coverage: a K percent statement coverage test is satisfied by any test data that results in the execution of K percent of the basic statements in the software program. Variations on statement coverage include such conditions as demonstrating that each executed statement is necessary. Another structural coverage measure is decision-to-decision branch coverage. This is a generalization of statement coverage. In statement coverage testing, a conditional or branching statement is considered the same as a non-branching statement: as long as it is executed by the test data, it contributes to the total coverage score of the test. For many programs that contain branches, however, the mere execution of a conditional statement is less significant than exercising all of the possible outcomes of the conditional. These decision-to-decision branches must be covered at the specified percentage in order to pass this kind of test. Coverage of higher level structural features may also be specified in a test plan. While structural coverage tests are seldom used as sufficient conditions for a test, they are commonly included in test plans as necessary tests.

Structural coverage testing is usually accomplished via the use of microscopic program instrumentation or, in other words, inserting recording or history collecting processes into the software. These routines do not affect the functional behavior of the system, but serve to provide key information for examination during test analysis and debugging activities. A research issue associated with this use of this technique is the performance implications which constrain its applicability when testing real-time software. Another use of program instrumentation is for the insertion of assertions into the software, so that the tester is notified whenever a basic assumption about the software's operating conditions is violated. Although automated tools may not be available to support microscopic program instrumentation in conjunction with Ada, the underlying technology is mature for non-performance oriented testing and should be easily extended to this arena.

It is common to use distinct processing paths in the software to partition the universe of possible inputs to the software into "domains." A frequently useful methodology can be based on simply specifying the proportion of the total number of paths that must be covered by the test data. This is not, however, a simple structural coverage test since these paths can involve the complex intertwining of structural features (e.g., the multiple and interleaved repetition of loops). Like the structurally-based tests, simple path coverage methodologies tend to be useful in generating necessary, but not always sufficient, tests.

Frequently, the domains have a mathematical or geometric structure that can be exploited. The "domain strategies" use previously gathered information about the likelihood of certain kinds of errors involving the definition of domains to generate tests. When depicted graphically, these tests might specify that a certain subset of the test points should be selected on one side of a domain boundary, a second subset on the other, and at least one test point should fall exactly on the domain boundary. If such rules guarantee that certain kinds of domain errors will be revealed (if present) then the domain strategy may be an effective necessary test.

One instance of this technique, partition analysis, requires the existence of a formal specification for the software and assumes that it is correct. Using this technique, the input domain is then divided into sets that are treated uniformly by both the specification and the software as implemented (Richardson and Clarke 1985). Research to date in this area appears promising; however, much work remains to be done if this is to be appropriate for application to large, complex software systems.

Tests that specify error coverage criteria are passed when test data are supplied that demonstrate that the given errors do not occur in the software. If the errors have a relationship to the test objectives, then error coverage tests are useful sufficient tests of some kinds of objectives. As a methodology for ruling out various kinds of errors, at least minimal error coverage tests are usually necessary. A common category of error coverage tests are mutation tests. These are the software equivalent of the single fault coverage tests frequently used for digital hardware. In mutation tests, software "faults" are modeled on errors that programmers are likely to make (DeMillo, Lipton, and Sayward 1978). Because of its statistical basis, mutation testing can be tailored to a given set of test objectives. Furthermore, it has been shown that by selectively applying mutation operators during the testing process, virtually all modern software testing strategies can be implemented (Walsh 1985). Efforts are currently underway to extend this technique for application to large Ada-based software systems (Appelbe et al. 1988).

Functional tests demonstrate the correct implementation of functional specifications and requirements. These may be either operational or technical specifications. While many functional testing methodologies are specific to the application at hand, several general concepts recur. A common goal of functional testing is to stress the functions. That is, to demonstrate the behavior of the software when limits and capacities are approached and exceeded. At the unit level, this may involve selecting tests that correspond to extreme or "out-of-spec" values for unit parameters or input variables. At the system or subsystem level, the corresponding stress test may involve test data that saturates a given system capability. While stress testing is seldom sufficient to determine whether or not objectives have been met, these functional tests are frequently necessary to exhibit the performance parameter limits of the system and to exhibit failure modes and effects.

Another useful functional test methodology is "random" testing. Random tests involve the selection, generation, or extraction of test data from a statistical or stochastic source. During unit testing, the statistical source may be a generator that samples from a sourçe according to a certain probability distribution. During integration testing, a simulator may provide the statistically meaningful frequencies of occurrence of data values, while during operational testing, the statistical variations in the actual mission profiles are the source of randomness. If the test designer has confidence in his knowledge of the underlying probability distributions, then random testing can be usėd to effectively estimate operational reliabilities and other statistical parameters (Duran and Ntafos 1981; Currit, Dyer, and Mills 1986). In these instances, random tests are used as sufficient conditions on the test objectives.

An operational test is another sort of functional test that explicitly addresses operational characteristics. Penetration testing is one type of operational testing that is of special interest to the security community. Operational tests are carried out in the operational environment using typical operator personnel. Although there are obvious analogies with "random" tests, operational tests are necessary to assess risk. For example, a system that cannot be sustained in a safe condition during an operational test is very unlikely to be worthy of deployment (DeMillo et al. 1987).

The extreme quality requirements of Ada-based high assurance systems result in the requirement to design and build testable software. This implies more than the need for modular, simple code. It implies that rather than depend on the insertion of probes at test time, ports must be designed and built into the software to allow visibility, as needed, during software execution. Macroscopic program instrumentation provides visibility to the tester in a form that allows the assimilation of information about the execution of large pieces of software. This capability is essential for high assurance software applications. During early test phases many properties of the running programs may be invisible to the tester without instrumenting devices such as counters that indicate which branches and paths have been executed. During tests of integrated software systems, it becomes even more difficult to peer into dynamic aspects of the software. In an integrated system running on an operational hardware set, it is usually very hard to directly observe such dynamic aspects of software performance without instruments that are tailored to the task. The alternatives are generally labor-intensive and are frequently not cost-effective to implement. For example, a common way of gaining visibility into such detailed features of the software is to interrupt an operational test so that the entire contents of the computer's memory can be "dumped" to magnetic tape or disk. This dump is then analyzed to extract information about the state of the software. Clearly this procedure cannot be used very often during the course of a test. On the other hand, it is frequently possible to place "software hooks" into delivered software so that hardware and software instruments can be conveniently attached during testing. It has been found that, when instrumentation requirements are recognized early in a program and are included in test plans and software requirements specifications, the quality of testing is improved.

Dynamic analysis has suffered in the past due to a reluctance to apply the requisite computing power. It is encouraging that recommendations are now being made to "exploit the advanced computer technologies to simplify the more difficult tasks such as software development and testing" (Eastport 1985, p. 15). Testing is an area that will benefit greatly from such a strategy. By cleverly combining modern testing techniques and advanced computer technology, it should be possible to thoroughly test the vast majority of Ada-based high assurance software applications expected in the future (Krauser and Mathur 1988).

3.2.3 Formal Verification

The goal of formal verification is to rigorously demonstrate, using mathematical logic, that software is consistent with its specifications. Techniques that have been applied to this problem include the use of input-output assertions and mathematical induction. Although the quest for a formal proof of the correctness of software may be appealing and research has progressed, some researchers have fundamental reservations about the method (DeMillo, Lipton, and Perlis 1979). Furthermore, although formal verification may achieve its goal, we still have no assurances that proven software will perform satisfactorily in its operational environment.

In spite of the limitations of this analysis method, formal verification research is being actively conducted. Notably, efforts to evaluate trusted computer systems strive for formal verification of designs and source code with respect to its specification (DoD 1983). Examples of technology efforts include the GYPSY project and the Vienna Development Method (VDM). Formal verification using Ada has been the subject of several workshops (Roby 1985).

3.2.4 Quality Assurance and Verification and Validation

Quality assurance and verification and validation are performed throughout the software development life cycle with the ultimate objective of building quality software. When properly applied, each activity has a distinct but complementary purpose (Fujii 1978). Quality assurance is concentrated on the definition of an appropriate approach to the creation of the software followed by continued monitoring of the development process for adherence. Verification and validation, on the other hand, is a continuous technical review of the software development products. (This use of the term "verification" should not be confused with Formal Verification.) In this context, verification is the process of reviewing the products of each step of the life cycle for consistency with the products of the previous phase. Validation is the process of testing the final software for satisfaction of the original requirements. Though each of these processes could benefit from additional automated support, the basic concepts are well established in practice.

3.2.5 Problem Reporting and Tracking

Approaches to problem reporting and automated systems for tracking the status of problem corrections are readily available. However, software with a requirement for fault-tolerance presents a unique need in this area. If a fault exists in the software such that fault-tolerance mechanisms are exercised, it would be desirable for the fault to be reported (e.g., through the use of an observation package to observe and record the event) even though the system failure was averted. In this manner, faults may be corrected as they are encountered, thereby reducing future stress on the fault-tolerance mechanisms (Choi et al. 1988).

3.2.6 Regression Testing

An important class of late life cycle tests is the regression testing that must be carried out to assess the impact of software changes. Key issues of regression tests revolve around the expense of re-running large numbers of individual tests. While these concerns are influenced by project management strategies, some technical aids are useful. Being able to assess the impact of program changes requires two-way traceability between program text and the historical record of test issues. Furthermore, errors, failures, and corrective actions should be reported and tracked to enhance traceability. Capabilities for effective and efficient regression testing are essential to the dependable operation of any software system. Although somewhat primitive systems do exist to control test information and support limited automated regression testing, the certain evolution and expected size of many systems make currently available alternatives inappropriate for the tasks at hand.

3.2.7 Quality Measurement

Control of a process implies the ability to specify quality requirements, predict success, track progress toward goals, and assess achievements. Today, software development is not a controlled process. However, this problem has been under investigation by the National Aeronautics and Space Administration (NASA), the Naval Weapons Center, the Rome Air Development Center (RADC), the Software Engineering Institute (SEI), and others for some time (Conte, Dunsmore, and Shen 86).

Software measurement and evaluation has been criticized for its lack of ties to basic scientific principles (Browne and Shaw 1981). Key omissions include the identification of invariant principles or relationships between basic measures, and hypothesis formulation and validation. Efforts to date have centered on modifying standard definitions into measurable software-related entities that bear little resemblance to those used throughout the remainder of the scientific community. One example is the following definition of maintainability: the ease of effort for locating and fixing a software failure within a specified time period (Bowen, Wigle, and Tsai 1985). A brief comparison of this definition with that provided in Section 2.5.2 reveals the omission of several key concepts, including the determination of a probability of restored functionality and the specification of the intended maintenance environment.

3.2.8 Reliability Modeling

An important but elusive goal is to faithfully model software reliability. A number of software reliability models have been proposed over the years. Most of these models borrow heavily from the field of hardware reliability and, as a result, the basic assumptions of the underlying probability distributions are to be questioned and carefully assessed prior to their use (Goel 1985, Rowland 1988). As opposed to efforts to model software reliability independent of the system, research is beginning to consider software reliability in the context of the system (Hecht and Hecht 1986).

3.3 FAULT TOLERANCE

Available technology for fault tolerance comprises methods and mechanisms to enable a system to continue operation in the presence of faults. Hardware fault-tolerant techniques are well established and form a vital part of any reliable computing system. The fault-tolerance strategies generally employed for hardware components rarely consider design faults. For software, however, design faults are a major concern.

Application of fault tolerant methods as a form of assurance technology requires system designers to:

- Identify components requiring fault-tolerance based on the criticality of the functions performed.
- Develop fault detection and recovery mechanisms.

Fault tolerance is a form of computing in the presence of noise. It is a basic theoretical limitation that the only way to assure correct computations is to build some form of redundancy into the system (Winograd 1962). In normal circumstances, redundancy is in the function domain (e.g., redundant components) or the time domain (e.g., roll back and restart), although other domains are possible. Whatever the approach, the following events appear to be necessary for effective fault tolerance:

- Detect faults and transition from normal to anomaly processing.
- Limit the extent of failure: isolate fault, limit fault propagation.
- Recover from failure: determine extent of failure, roll back to consistent state, propagate failure reports.
- Prevent recurrence: trace cause of failure, reconfigure system components.

3.3.1 Identification of Critical Components

A technique for identifying critical components of a system is *fault tree analysis*, which was adapted for use with software by Leveson (Leveson and Harvey 1983). This techniques starts by assuming that a critical failure can occur, which is called a *loss event*. The sources of failure are then derived by working backwards from the loss event. Each contributing failure may have several sources, producing a tree structure of possible faults. This analysis is continued until the leaves of the tree represent individual software modules. This process, therefore, can identify conditions, components, and design shortcomings that can lead to critical failures.

3.3.2 Fault Detection

Redundant software is one approach to fault-detection. This method is similar to the hardware technique of active redundancy, which employs multiple units to perform identical functions in parallel. In hardware systems, arbitration of conflicting results is handled by "voting" circuitry. In software, additional statements are required to (1) determine if agreement has been reached and (2) determine which result to honor in the event of a conflict.

Assertion monitoring provides an explicit check for the plausibility of results or system states. This check is performed during program execution and is used to detect faults without redundancy. This method may have some advantage over multi-version voting techniques. (Leveson and Shimeall 1983).

N-version programming is another approach to fault-detection that requires several independent versions of a piece of software be produced for a specific application, usually from the same specification. These versions are then executed in parallel and the outputs are subjected to a majority voting process. The voting process determines the results which will be carried forward for use by the remaining components of the system. Maximum utility is provided when identical requirements can be stated in highly dissimilar terms. Since developers often make similar mistakes, researchers have suggested the need for additional research to identify common errors that result in software faults (Avizienis 1976; Knight, Leveson, and St. Jean 1985).

3.3.3 Fault Isolation

Once it has been determined that a fault has occurred, it is imperative to confine the propagation of the anomalies at that level of operation. The propagation of subsequent faults will further degrade the performance, reliability, and survivability of the system. Modular organization of software is a demonstrated asset to fault-recovery (Scott et al. 1984). It is also particularly beneficial in limiting the extent of failure propagation (Krishna, Shin, and Butler 1984).

3.3.4 Recovery Mechanisms

Exceptions and exception handling provide a mechanism for transitioning within a fault-tolerant framework. This framework provides a clear separation between normal and abnormal activities. The occurrence of a fault (detection of an error) results in the raising of an exception. The handler associated with that exception will be evoked and some fault handling activity appropriate to the situation will be initiated. Some research (Goodenough 1975) has suggested the use of exceptions for general processing, others (Liskov and Snyder 1979; Cristian 1980) have proposed reserving the use of exceptions to the mechanism of fault-tolerance.

Recovery blocks (Avizienis 1976) incorporate error detection, backward error recovery, and fault treatment. The implementation of a backward recovery mechanism as a part of this framework precludes the need for a damage assessment strategy. The backward error recovery mechanism assumes the elimination of all damage caused by a faulty module. This technique provides a coherent framework for encapsulating redundancy in a software system (Leveson and Harvey 1983).

3.4 ACQUISITION AND MANAGEMENT PRACTICE

All of the software technology reviewed previously is employed in the context of some organization's approach to doing business. On Ada-based software system developments, the approach is usually a combination of contractually imposed customer standards and the contractor's internal standards and procedures. This section will examine the influence that management practices exert on technology and vice versa.

3.4.1 Life Cycle Models

The relationships between the software development activities of specifying, designing, coding, testing, and maintaining the software are defined by a variety of life cycle models. It is clear that virtually all software systems evolve. However, there is no single approach to the evolution of software. The choice remains as to whether the evolution will occur as the waterfall model repeated sequentially, for example, or continuous multiple parallel developments. Other models cycle through the specification phase until satisfactory results are achieved, then design and build pieces of the system. Another variation is to completely specify and design the system, and then build and test in smaller pieces. As with the majority of the software technology discussed, there is little to no objective evidence available to aid in the choice of a development model whose use will result in high quality software for a given project. The selection must be guided by engineering judgement and the expected timing of the availability of information about the system requirements (Martin 1987).

3.4.2 Organizational Approaches

The involvement of multiple independent groups is important to the development of high assurance software. This is necessary if misconceptions about the operational environment and misinterpretations of requirements are to be avoided. Therefore, commonly employed techniques include the use of independent test teams, independent quality assurance groups, and independent verification and validation organizations. In cases where N-version programming is being used to build fault-tolerance into the system, multiple independent development groups are employed. The basic principle being followed is to minimize the influence of individuals or single organizations on the ultimate quality of the system. In addition to the organizational independence, the evolution of modern systems may force the use of parallel development groups. Multiple concurrent developments are not uncommon in evolutionary systems. The required independence and parallelism in development organizations, as well as the size of many development projects, create extreme demands for effective management control and careful coordination.

3.4.3 Configuration Management

The complications that arise due to the choice of life cycle model and organizational approach demand the disciplined application of a sophisticated configuration management system. The only scenario in which configuration management is not a concern is that where error-free software is produced by a single person to satisfy a requirement that will never change. This is not the expected scenario for the development of Ada-based high assurance software. Although a number of configuration management systems are commercially available, their capabilities need to be carefully examined to determine if they support Ada-specific system structures, systems with large numbers of components, and evolutionary models that incorporate multiple, concurrent developments of individual system functions.

3.4.4 Effect of Standards and Quality Resources

Examination of past programs has determined that the primary influence on the choice of software development technology has been the standards and requirements as defined by contractual agreements (Redwine et al. 1984). This implies that conscious, careful acquisition and management decisions must be made with respect to the desired software technology.

Ideally, every project should employ the most qualified and talented personnel available. Few projects are fortunate enough to achieve this goal. Instead, a software development team is composed of a mixture of talented, experienced individuals, conscientious but less-skilled persons, and unknown quantities that are just entering the work force. The equalizing factors that can be applied to balance the risk of this less than ideal situation are technology and training.

Computer power should be used to the maximum extent possible to alleviate the mundane demands on personnel resources. Prior to relying on support software for the injection of some degree of quality in a software system though, confidence in the support software itself must be justified. In particular, testing tools and compilers must be tested and validated to ensure the advertised functions are in fact being effectively performed.

3.5 SUMMARY

The development of correct and accurate specifications is an essential prerequisite to the development of high assurance software. There is virtually no chance of success on large projects without a good set of specifications to drive all development activities. Examples of promising specification technologies include prototyping, executable specifications, and computing-oriented specification languages and processors. Also necessary is specification technology that will allow the definition of quality and assurance requirements in operational terms.

The only way to gain confidence that the software will perform as intended in the operational environment is through the use of systematic dynamic analysis techniques. Some of the promising techniques stress the observation of software performance under varied conditions. These include program mutation techniques and macroscopic program instrumentation. Other techniques emphasize the determination of a minimal, adequate set of test data. Software test data generation techniques in need of further development include symbolic execution and input space partitioning.

Virtually no research exists on operational testing of software. Appropriate definitions and models have yet to be formulated. Systematic techniques for operational testing in such areas such as real-time, parallel, and secure systems .re, for the most part, lacking. Developing a technology base in these areas should be given the highest priority.

Selected software testing techniques are mature enough that extensive investment in research activities is not needed to gain the benefits of their utilization for Ada-based high assurance systems. The needs of these techniques lie in the area of technology modification and transition, specifically to the arena of Ada-based systems. In some cases, further modification may be necessary to scale-up the capabilities for application to large systems. Examples of testing techniques appropriate for transition consideration include static analysis, functional testing, random testing, and regression testing.

Quantitative models and measurement techniques are needed to provide precise methodologies for requirements specification, determining system development status, and predicting future performance of the system. Research in this area has almost exclusively centered on simple adaptations of existing statistical models, statistical data analysis, small-scale experimentation, and the compiling of anecdotal evidence. Research is needed in approaches that have deeper scientific roots. The development of mathematical cause-effect models and execution of large-scale experiments to validate model predictions are essential to progress in this area.

Finally, quantitative and reliable cost-benefit data are almost completely lacking in the software engineering community. Project managers must either believe marketing claims of vendors or conduct original research to extract cost-benefit trade-offs from analytical studies. Neither approach is satisfactory for Ada-based high assurance systems. There is a critical need for demonstration experiments structured to provide quantitative indicators of risks incurred by: (a) selection of a given technology, (b) selection of an alternative technology, and (c) use of ad hoc methodologies. These indicators when coupled with precise estimates of costs would provide sufficient data for rational decision making when structuring a software development program.

4. ASSURANCE REQUIREMENTS

Of critical concern is how and when to set assurance requiremen'ts and how to determine when they have been satisfied. This section outlines a process for:

- Identifying the system level risk drivers associated with high assurance software
- Refining the required characteristics for critical software functions
- Defining the critical issues to be addressed by the assurance program
- Establishing the assurance program to address those issues
- Setting software assurance requirements
- Applying assurance methods

4.1 DERIVING ASSURANCE REQUIREMENTS

Assurance requirements will arise from two mechanisms:

- Derived assurance requirements: these are obtained by risk analysis of the system and are, for example, typically related to the cost of failure of critical system functions.
- Imposed assurance requirements: these usually result from special certification (e.g., flight critical software) or safety needs (e.g., nuclear) of the system.

Whereas derived requirements are used to ensure a specified level of risk in the system, imposed requirements are more useful for regulatory purposes—to ensure a specified level of assurance on all systems of a given type, for example. In either case the overall goal is to balance risk.

4.2 SYSTEM LEVEL SOFTWARE RISK DRIVERS

4.2.1 Critical Functions

A critical function is any system function without which the system would fail to satisfy some or all of its operational objectives. The prime concern in systems that implement some or all of their critical functions in software is whether or not the software has been given a balanced treatment with other system components. Experience has shown that when these considerations are pushed later into the development process, latent problems with the software are more difficult to eliminate and the resulting systems are less well-suited to their objectives.

To emphasize the balance that should be sought between the software and the hardware components that implement critical functions, the term critical software component is used. A critical software component fulfills a requirement for a key function.

4.2.2 Mission/Function Matrix

A matrix relating operational requirements of the system to system functions is the primary source of information about how the capabilities have been partitioned between hardware and software. These partitions will be important in determining required characteristics, in defining error or failure categories and in isolating and correcting deficiencies noted during the remainder of the test program. It is especially important that proper engineering studies have led to the establishment of these partitions.

4.2.3 Identifying Risk in Software Functions

Operational requirements reside in the operational concept descriptions of the new system. System functions are designed to meet the operational objectives. An understanding of the sources of risk in each of the software implemented functions thus identified is an important part of overall system risk assessment.

New software functions represent the highest risk, since they involve not only the design of the software but also the use of new concepts, theories and algorithms. Often, these functions have only been demonstrated in laboratory or experimental setting and no operator personnel have been exposed to the functions under realistic conditions. Questions of suitability (e.g., availability, reliability, safety) are typical for these functions and the early involvement of users and operational testers is encouraged. Risk reduction procedures such as prototyping, simulation and evolutionary acquisition may also be appropriate.

The transition from manual functions to automated functions is notoriously hard to manage and assure. Functions performed by humans are usually difficult to specify. It is therefore hard to test conformance of the automated capability to a fixed technical specification. On the other hand, the functions themselves are usually mature, so suitability risks may be somewhat reduced. In either case, there should be a clear plan for determining the extent to which the previously manual capability has been faithfully reproduced.

Re-use of mature software (i.e., existing software that has been extensively used without failure or other defect for a significant period of time) or reimplementation of well-understood functions represents relatively lower risk. However, for a system with extreme reliability requirements the extent to which the re-used components have been subjected to rigorous test and evaluation may present special risks.

Electronic interfaces are responsible for interoperability, communication (in the underlying network) and man-machine communication. These interfaces are frequently software intensive. Even if the hardware and much of the software is off-the-shelf, much of it will be developed under the development program. Furthermore, many aspects of the software architecture will depend upon and anticipate functions in interfaces. Such issues as early estimation of performance impact of interfaces, isolation and tolerance of system functions from interface faults, adequacy of the man-machine interface and the effect of the various interoperability requirements on the overall system design are critical.

4.2.4 Other Risk Drivers

The use of Ada in current systems is a significant source of risk. Support environments are only beginning to mature and the reliability of Ada compilers has not yet been tested to any useful extent.

A common source of operational software problems is the difficulty of maintaining and supporting the software once it is deployed. Common danger signals include the use of proprietary design tools and methods that will not be available to maintenance organizations. The lack of experience with Ada is a significant source of risk here.

4.3 REQUIRED CHARACTERISTICS

4.3.1 Goals and Thresholds

Most operational requirements for availability and maintainability are expressed in terms of goals. A necessary component of system level test planning is the definition of goals and thresholds for the critical software components. The threshold requirements represent the minimum levels of operational suitability (e.g., availability) and effectiveness (e.g., maximum workload at an operator's station) below which the system will be not deployed. Operational goals and thresholds are important for setting design criteria and priorities. Operational thresholds are critical for testing.

Without a definition of threshold requirements, the system is at risk during the acquisition phase. It is commonly recognized that extreme requirements will not be met in the initial releases of the systems but rather will be achieved during reliability growth programs. The assurance plan should then specify the range of acceptable system reliability. In other words, having failed to achieve the goal, by what criteria is the system to be evaluated and deployed?

4.3.2 Software Specific Characteristics

Special care should be taken to ensure that required software characteristics have been identified. Software characteristics should be evaluated at the appropriate stage of system development rather than at arbitrarily imposed milestones. It is, in general, a mistake to wait until hardware and software are integrated to resolved outstanding software test issues.

Late evaluation of software characteristics opens the following problems:

- *Error masking*: hardware and software errors may in some instances mask each other, making reliability, availability, and maintainability (RAM) analyses impossible.
- *Error partitioning*: without a reliable estimate of software failure rates, the partitioning of test events into hardware, operator, and software failures will be subjective and inexact.

These problems would, of course, amplify the already formidable problems of evaluating complex high assurance systems.

4.3.3 Critical Assurance Issues

A critical issue is any aspect of a software system's capability that must be questioned before the system's overall worth can be estimated. The software issues are of primary importance in reaching a decision concerning whether or not the acquisition should progress into later programmatic phases. This decision should be based in part on the determination that the goals and thresholds defined for the required software characteristics have been met and, in any case, should be based on an assessment that software and hardware risk have been balanced by past and future test and evaluation.

Regardless of assurance approach, the critical issues should address the high-risk nature of the software aspects of the system.

4.4 ADEQUACY OF ASSURANCE PROGRAMS

4.4.1 Management

The following procedures have been established in DoD Directive 5000.3, "Test and Evaluation," for judging the adequacy of assurance program management. Organizational concerns will of course be tailored to the special concerns of the system and some organizational mechanisms may not be appropriate in all instances. However, the guidelines apply in all circumstances:

- Early identification of independent development and operational test organizations

- Early involvement of independent testers in requirements definition and analysis
- Clearly defined decision paths for test and evaluation
- Effective independent verification and validation (IV&V) that highlights and prioritizes outstanding or unresolved critical issues

The assurance program itself should be integrated with the design and acquisition of the system. A total program of risk reduction through incremental verification and validation represents the best structure for managing and evaluating the status of the software as it evolves into a deployable system.

4.4.2 Integrated Schedule

The integrated schedule displays the time sequencing of test and evaluation for the entire program and related key events in the acquisition decision-making process. Included are such events as program decision milestones, key subsystem demonstrations, test article availability, critical support resource availability, critical full-up system demonstrations, key operational test events, first production deliveries, and initial operational capability date.

The schedule should also include such events as key software subsystem demonstrations and software test article availability. The schedule should include adequate allowance for repair and retest of software, as well as time to perform the original tests.

Support resource availability should be displayed in the schedule. Software assurance tools fall into this category and deserve special mention. Since these tools are themselves software, their development and acquisition are subject to the same risks as any other software development. The availability of these tools should be included in early test planning and tracked by the test program management since a late delivery could impact the entire test program.

4.5 SOFTWARE ASSURANCE REQUIREMENTS

4.5.1 Demonstrating Required Characteristics

Assurance technologies should be designed to demonstrate required characteristics. In particular, the aim of development testing is to ensure that the phases of system development result in intermediate "validation products"(see Figure 2). These are used to demonstrate the three kinds of consistency that affect the flow of requirements and specifications during system development:

- Propagated consistency: at each increment of the development process,

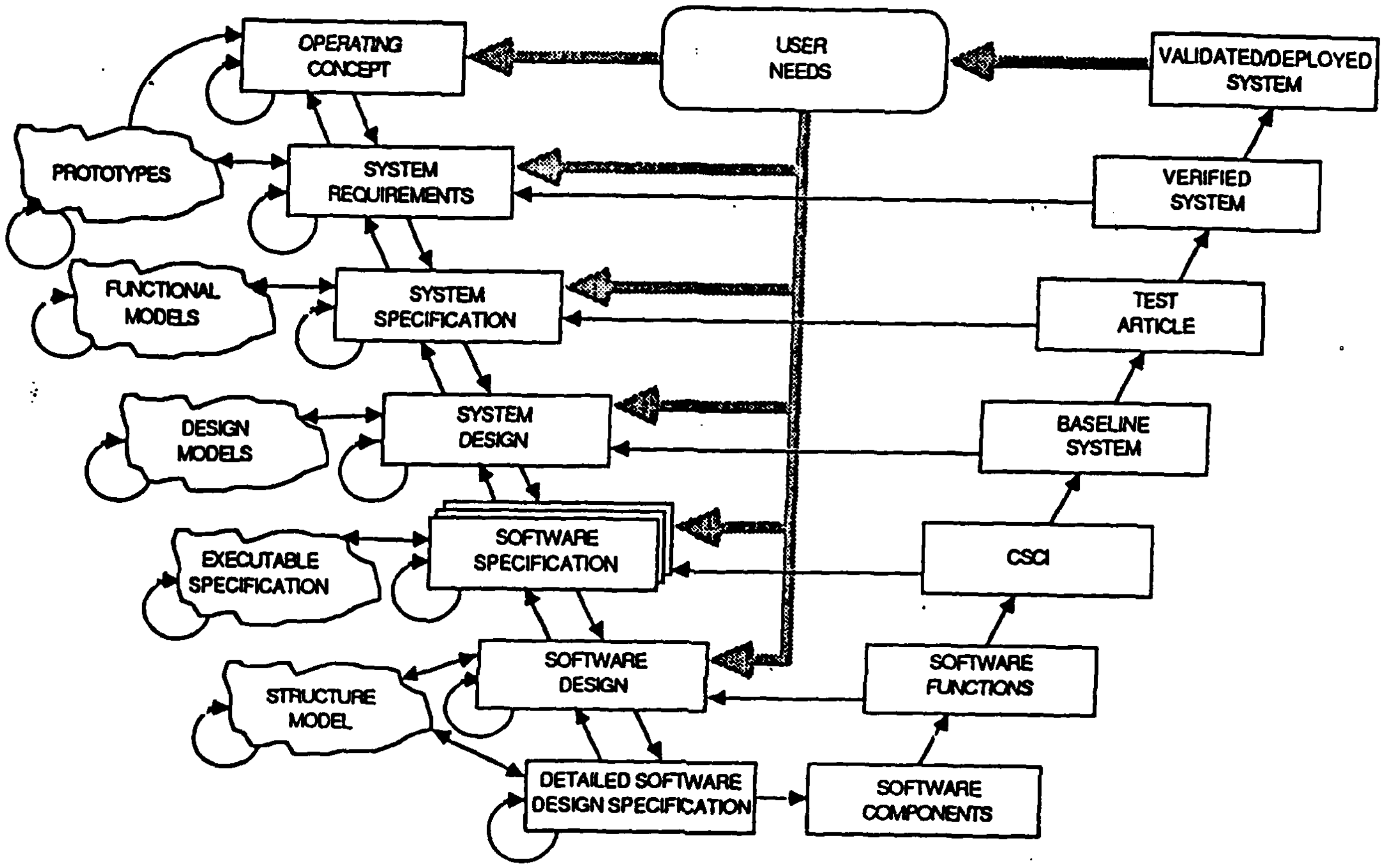


Figure 2. Incremental Verification and Validation Process

validation products should be consistent with their immediate predecessors (e.g., software designs should be consistent with software specifications).

- Self-consistency: each validation product should be free of internal contradictions (e.g., the system design should be self-consistent).
- Consistency within increments: at each stage of integration, the product of the current increment should be consistent with the corresponding validation product (e.g., the baseline system should be consistent with the system design).

4.5.2 Validation Products

A validation product is any engineering product whose primary purpose is validation of user requirements. Unlike auxiliary assurance products like test reports, a validation product represents the system during an assurance activity. Assurance products are shown outlined in irregular bubbles at the left hand side of Figure 2. Notice that, since each validation product is itself a surrogate for the system, care is taken to ensure consistency between the validation products and the corresponding intermediate system design products (e.g., executable specifications must be consistent with software specifications).

To the maximum extent possible, the validation products cited in Figure 2 should lend themselves to early operational demonstrations of key functions and capabilities. As described above, demonstrations that occur after hardware-software integration tend to obscure deficiencies that involve hardware-software interactions. In general, prototype demonstrations using typical operator and maintenance personnel are preferred during early development phases.

Development testing is the latest stage at which individual software modules are available for direct stimulation and manipulation. This makes software development testing a critical element of the overall test program. Most systematic testing that takes place during this phase attempts to cover, stress or overload software functions and parameters. This is a critical concept since after software modules are integrated and linked, it will not, in general, be possible to address such test issues. More specifically, if a particular fault-tolerant module M is supposed to detect and recover from fault events A, B, and C, it is desirable to exercise the module in the presence of the events A, B, C, AA, AB, AC, BB, BC,..., ABC, and so on. The events in question may in turn be triggered only in the presence of data items that exceed specified range values. While it is generally possible to use test drivers to supply M the required values, external manipulation of those values by systems test methods is frequently not possible.

4.5.3 Handling Deficiencies

The relationships between test events, software deficiencies, and unresolved test issues are more difficult to discover the later in the acquisition process they arise. Relating software test results to system-oriented test issues helps to ensure that responsibilities for deficiencies are properly allocated between hardware and software, and the test plan should include the necessary analysis for this activity. Early development testing is still the best opportunity for tracing system characteristics to design characteristics.

Test deficiencies should be traced to the corresponding required characteristics. In particular, vague references during contractor-conducted testing to "successful software demonstrations" or "no problems with the software" should not be acceptable. By the same token, jargon is especially difficult to interpret in test reports. Phrases like "buffer overflow causing module JXAS115 to hang" can camouflage software deficiencies. At each review phase, the essential questions are:

- Were the development assurance objectives met?
- With what degree of confidence?
- What behaviors led to the observed anomalies?

Partitioning of hardware and software deficiencies is most easily carried out during early development testing. Plans should specify methods for deriving early software test articles in order to accomplish this partitioning.

4.6 APPLYING ASSURANCE TECHNOLOGY

This section presents a guideline for planning and conducting software assurance activities. General approaches to structuring and monitoring the assurance program have been defined and discussed above.

There is relatively little experience with high assurance software systems (i.e., with testing systems having extreme reliability requirements). It is certain, however, that the problems encountered in testing these will be at least as hard as those encountered in tests of other software intensive systems. On the other hand, there is reason for optimism about the applicability of emerging assurance technologies:

- Available assurance methodologies are not sensitive to the application domain. That is, the assurance methodologies that are available are valid for any software system, including those with high assurance requirements.
- Hazard types are generic. The software risk drivers that will be addressed during the assurance program are defined in terms of generic software concepts, not specific architectural properties. The unique aspects of high assurance lie

in the high coverage requirements for these hazards rather than in the kinds of hazards.

4.6.1 Identify Critical Functions and Components

As described above, not all software components present the same levels of risk. Software functions should be partitioned and related to critical operational requirements. The system operating concept provides a coarse guideline for carrying out this partitioning. Critical functions fall into one of two operating categories:

- Functions that provide for transition (e.g., from hot standby to standoff or to an operating mode) are the most critical in the system since the system function cannot be sustained during catastrophic failure without these functions.
- Emergency functions that must be sustained over specified periods if the system function is to be carried out under expected workload levels.

4.6.2 Determine Thresholds and Assurance Issues

Each of the operational requirements should be accompanied by a threshold requirement—that is, a floor above which the system must be sustained if safe services are to be provided within the specified performance envelope. Thresholds are established for critical software functions by tracing them to corresponding thresholds in the operational requirements/system function matrix. For example a threshold requirement for availability of at least one backup for a specified set of functions can be transformed into thresholds on the availability of the functions that transfer from an operating mode to the backup.

Application-dependent and architecture-dependent information are used to determine critical assurance issues for testing.

These issues are usually expressed as special risk factors or hazard classes that, unless removed or compensated for by architectural means, will cause the function to fall below threshold requirements and to therefore be unsuitable for deployment. Some typical issues include the following:

Containment of Non-Development Software

It cannot be assumed that risk reduction will have taken place on non-development (third party) software. It is probably the most severe risk driver. Even the most rigorous fault-tolerant mechanisms cannot cope adequately with these faults since the non-development items have unknown and in many cases undefined failure modes and effects.

Timing and Synchronization

Real time design has been highlighted on many previous occasions as a principal risk driver. Without careful incremental testing at the unit, module, and later levels, it is nearly impossible to verify coverage of these faults with any useful degree of precision.

Load and Stress

These are hazards that result from loading operational parameters at or near specified limit values. These loads stress internal parameters which then interact with system functions in unpredictable ways. For example, operational loads on a operator's position may result in the overflow of internal tables that result in loss of data in track files (a common failure mode in radar systems). Generating these loads during operational, acceptance or system tests is generally not feasible. Without incremental assurance at the unit, module, subsystem, integration and hardware-software integration levels, it is nearly impossible to verify coverage for these faults with any useful degree of precision.

Hazards in Assurance-Enhancement Mechanisms

The criticality of these as issues depends to a large extent on architectural considerations. If, for example, a critical function does not depend on tolerating hardware and software faults, then faults in the fault tolerant mechanisms may be less important. Nevertheless, risk drivers in the fault-tolerant parts of the software are likely to be abundant.

As the complexity of the fault-tolerant mechanisms increases so does the risk associated with uncovered faults in those mechanisms. Recovery blocks, coordinating software for redundant components, votes, and roll-back mechanisms all contain such complex mechanisms. Coverage verification requires an independent but equivalent fault model implemented in a suitable test environment and is therefore difficult to achieve.

Latent or Residual Faults

These are faults that, by virtue of their placement in the software (relative to the distribution of input values), or factors indigenous to the application, acquisition strategy, or contractor's software development practices are likely to remain in the software at any specified point in time. This class includes latent design faults and defects such as those that might arise from incorrect design specifications, incomplete requirements or improper documentation. Without careful incremental testing at all levels, it is nearly impossible to verify coverage for these faults with any useful degree of precision.

Conducting these tests in a controlled and incremental fashion as shown in Figure 2 is absolutely necessary to the success of the overall test program. The principal reasons for emphasizing the incremental approach are the following:

- Many critical issues can only be addressed by incremental application of fixed risk reduction mechanisms.
- The cost of system level application of many required assurance methods is prohibitive unless the methods have first been applied at lower levels of integration.
- Early validation products and subsystem demonstrations allow early operational assessments of suitability.
- The cost of locating and correcting software faults increases by a factor of 100 or more as the the system is integrated.

In assessing the assurance technologies to be used, it is important to select those that:

- Can be applied incrementally at all stages of development from requirements modeling to system testing
- Are designed to address the most likely critical issues

In many coverage methods, the expense increases non-linearly as component size increases. It is therefore *required* that these tests be applied in incremental fashion to keep testing costs manageable.

The validation products mentioned in Figure 2 must be viewed as engineering products. They should be specified in statements of work and the integrated schedules should account for their development. Applicable standards should be interpreted to require such products.

There is at least one school of thought in the design community that does not include individual software units as validation products. Avoiding or delaying access to units for purposes of assurance always increases risk.

4.6.3 Determine Assurance Objectives

The critical issues should also determine a quantitative test objective for each test. This distinguishes the systematic approach to software testing from less rigorous and less effective "test-a-lot" approaches. It is essential that each test that is planned and conducted be accompanied by a clear set of criteria for success or failure. Failure to meet a quantified test objective must be treated as a deficiency to be resol"ed by a later critical test issue.

4.6.4 Coverage of Hazards

Structural Coverage

Structural coverage tests exercise every component (e.g., statements, decision-to-decision branches) of critical functions. High assurance requires high structural coverage. For example, every statement of a program must be executed to demonstrate that it is free of statement-level faults. Any set of tests that do not execute every statement at least once fail to achieve structural coverage at the statement level.

Domain Coverage

Domain coverage partitions actual component behaviors or specified program behaviors into classes. These classes may relate to safety states (e.g., safe vs. unsafe) or to finer-grained distinctions such as whether or not a given function is performed. The test strategies surrounding domain coverage are used to address the following test issues:

- Presence of timing and synchronization faults: the domains represent event sequences that induce synchronization dependent information (e.g., whetner or not a pair of functions is driven to deadlock).
- Latent or residual faults: the relationship between specified and implemented domains can be systematically explored for "what-if" analysis and to determine missing paths and conditions.

Faults and Errors

These tests address all of the issues for which a precise fault model has been identified. The tests then determine that, in the presence of a specified fault, the implemented software component behaves as specified or not. If the faulted component behaves differently than the original, then the test has either uncovered an error in the original or has demonstrated that the original does not contain that specific error. This method, sometimes called mutation analysis, can be automated.

Functional Models

Functional tests use properties of the underlying application specific model to construct tests that give a high degree of confidence that the model has been satisfied. Suppose, for example that two digital filters F and G implement polynomial transformations on their inputs $X_1, \ldots, X_n$. Then there are exact tests to determine whether or not $F(X_1, \ldots, X_n) \equiv G(X_1, \ldots, X_n)$. Other functional tests use random selection of input values chosen according to specified distributions to estimate operational reliability. An important category of functional tests includes those operational tests that are conducted with early validation products.

Interface Functions

Functional tests that stress or exercise interfaces between functions (particularly those used in combination with error-based tests) are the only way to address some fault classes such as load and stress faults and faults in nondevelopment software.

For example, tolerance of fault in nondevelopment software is tested by developing a fault model of interaction between development and nondevelopment software and simulating the faults at the interfaces. By the same token, stress faults are tested by deliberately driving parameters and constants outside specified envelopes.

5. RECOMMENDED STRATEGY

5.1 ASSURANCE REQUIREMENTS

Recommendation: Review and, if necessary, revise existing acquisition policy and implementing materials to ensure that the incremental assurance process is institutionalized.

As described in Sections 2 and 4, the refinement of required characteristics from operational system level requirements to technical code level requirements and the incremental execution of an assurance plan is necessary to high assurance systems. The flow of requirements should be reflected in policy and its implementation. This places ultimate responsibility with the acquisition decision-making chain. This is appropriate since the decision chain is the authority for all risk-related acquisition matters. Among other things, this approach encourages software assurance requirements to be established in order to balance hardware and software risks.

Institutionalizing high assurance methods in the acquisition process also helps to guarantee a measure of coordination since policy directives can call for uniform implementing regulations, standards and technology.

5.2 STANDARDS

Recommendation: Review and revise DoD-STD-2167A and DoD-STD-2168 to require the application of appropriate assurance methods to high assurance systems.

Since much of the operational risk of a new system is born by its users, it is incumbent upon contracting officials to place firm safeguards in statements of work and contracts. If assurance technology is not required by standards and regulations that are called out in contracts, there are virtually no acceptable alternative mechanisms. Both the Software Development Standard and the Software Quality Standard should incorporate requirements for incremental assurance programs. Furthermore, these standards should be key implementing documents for higher level policy and guidance and should therefore be reviewed in that light.

Recommendation: Develop a standard software engineering glossary that is acceptable to all DoD components listing terms, definitions, and concepts that conform to standard engineering usages.

As noted above, the proliferation of ad hoc and technically unsound definitions of standard terms like "reliability" is detrimental to the development of sound, uniform *approaches to high assurance systems.*

5.3 TECHNOLOGY INSERTION

Recommendation: Use existing technology insertion mechanisms for assurance technology. Incentivize contractors to develop and deliver productized versions of assurance tools.

DoD software initiative organizations already exist to facilitate technology transition and insertion. Among the most important of these are the Ada Joint Program Office (AJPO), the Software Engineering Institute (SEI), and the Software Technology for Adaptable and Reliable Systems (STARS) program. Thus far, these organizations have shown little interest in investing in assurance technology. Special programs with legitimate interests in high assurance software—such as the Strategic Defense Initiative Office (SDIO)—have also expended little effort in this area to date.

On the other hand, agencies such as the National Computer Security Center (NCSC) have repeatedly demonstrated how to insert assurance technology through a program of institutionalizing assurance requirements and selectively funding productization of promising technology. While the Security Center has concentrated its efforts on one technology (formal verification) because of its mission, organizations with less restrictive charters can adopt similar methods to broaden support for other promising tools and techniques. By the same token, the NCSC can expand its role in fostering high assurance technologies other than formal verification.

Explicit assurance requirements in contracts coupled with an economically realistic acquisition strategy for assurance technology productization provide an adequate set of incentives.

In effect, the existing mechanisms are probably adequate for beginning the insertion process. However, if arbitrarily imposed schedule and budget constraints remain the highest priority in acquisition, then these insertion mechanisms are certainly not adequate.

5.4 REQUIRED RESEARCH

Recommendation: Establish a broadly-based Tri-Service research initiative to rebuild the research community in software testing, analysis, and verification. This program should be coordinated with the National Science Foundation (NSF) and other government agencies and should have a minimum goal of increasing funding four fold over a five year period.

The research community in assurance technology is in a shambles. Federal funding for research in this area is either flat or declining. In 1979, the principle technical symposium in assurance technology drew over 400 participants. In 1986, the symposium attracted less than 80. The little industrial research that is conducted is not directed to DoD problems. It is, for the most part, neither funded nor managed by DoD components. There are few university centers of excellence in assurance research. In contrast to the early 1970's, none of the top five university computer science departments in the US has a critical mass of researchers in assurance technology. Only one academic computer science department in the top 10 has an active research group. On almost every front, research in assurance technology has declined since 1976.

Recommendation: Direct DoD Scientific Research Offices (Air Force Office of Scientific Research (AFOSR), Army Research Office (ARO), Office of Naval Research (ONR)) and the Defense Advanced Research Projects Agency (DARPA) to fund both innovative assurance technology projects (6.1) and longer-term projects (6.2) aimed at maturing and validating existing methods.

It should be clear that there are no magic solutions to the high assurance system problem. Advances will only be made by persistent development of promising methods. The following should be the highest priority:

- Continued and deeper basic research on dynamic testing strategies
- Support for laboratory and experimental tool development to help determine cost-benefit tradeoffs for dynamic testing methods
- Development of models and methods for testing real-time embedded software
- Development of methods and models for operational testing of mission critical software
- Full scale experimentation to determine the scope and applicability of formal program verification for high assurance software
- Full scale field trials to facilitate transition of technology
- Gathering and analysis of software error data to help in determining hazards to high assurance software development

This will require not only the funding of single investigator research projects aimed at innovative solutions but also the funding of long-term multi-investigator efforts aimed at large scale experimentation and field data gathering. The long term goal should be to achieve a coverage of approaches and in-depth studies of selected approaches.

Recommendation: Develop joint industry-university research initiatives.

Universities and industrial R&D centers are beginning to embark on cooperative ventures aimed at commercial problems. The advantages for universities is the availability of long-term collaboration and support from industrial partners (who may provide, for example, field data for validation studies, state-of-the-art equipment, facilities for productization, etc.). The advantages for industry include early access to research results that provide a competitive edge, contact with students and faculty, and leverage for their research investment. DoD interests in these joint ventures are only sporadically represented. The result may be that new technology is employed for exclusively commercial purposes, further slowing the insertion of that technology for DoD problems.

5.5 COORDINATION

Recommendation: Implement the recommendations identified in Sections 5.1-5.3 in existing software technology offices through the Director, Defense Research and Engineering (DDR&E) and his Deputy for Research and Advanced Technology (DDDRE(R&AT)).

Even though development organizations contain concentrations of expertise in assurance technology, their near-term horizons limit the possibilities for coordination.

Recommendation: Coordinate with the test and evaluation community.

Broad-based approaches to assurance cannot be successful without the involvement of the Test and Evaluation offices within the Office of the Secretary of Defense (OSD)— the Director, Operational Test and Evaluation (DOT&E) and the Deputy Director, Defense Research and Engineering for Test and Evaluation (DDDRE(T&E))— and in the Services (e.g., the Air Force's Operational Test and Evaluation Center (AFOTEC) and Director of Test and Evaluation (SAF/AQV); the Navy's Operational Test and Evaluation Force (OPTFOR) and Director of Research and Development, Test and Evaluation; and the Army's Operation Test and Evaluation Agency (OTEA) and Test and Evaluation Command (TECOM)). These offices are responsible for testing and evaluation of acquisition risks for all major defense systems. Immediate steps should be taken to determine the software technology requirements imposed by existing and planned T&E guidance. Furthermore, future R&D in high assurance software should be planned to complement the thrusts and initiatives of the T&E community.

In addition, existing T&E models, definitions and system acquisition concepts should be integrated with high assurance software technology development, transition, and insertion programs.

Recommendation: Institutionalize changes.

It is a unique aspect of assurance technology that institutionalizing a sound process for applying it is enough to ensure coordination.

REFERENCES

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. 1985. *Awk—A pattern scanning and processing language programmer's manual.* Murray Hill, NJ: AT&T Bell Laboratories. Computing Science Technical Report No. 118.

Alford, M. 1977. "A requirements engineering methodology for real-time processing requirements." *IEEE Transactions on Software Engineering*, SE-3, 1 (January): 60-69.

Appelbe, W. F., R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. 1988. *Using mutation analysis for testing Ada programs.* West Lafayette, IN: Software Engineering Research Center, Purdue University. SERC-TR-9-P.

Avizienis, A. 1976. "Fault-tolerant systems." *IEEE Transactions on Computers*, C-25, 12 (December): 1304-1312.

Balzer, R. 1981. *Gist Final Report.* Information Sciences Institute, University of Southern California.

Balzer, R. M., N. Goldman, and D. Wile. 1976. "On the transformational implementation approach to programming." In *Second international conference on software engineering*, San Francisco, October 13-15, 1976: 337-344.

Barr, Avron, and Edward A. Feigenbaum, editors. 1982. *The handbook of artificial intelligence: Volume 2.* Los Altos, CA: William Kaufman, Inc.

Bentley, Jon. 1986. *Programming pearls.* Reading, MA: Addison-Wesley.

Bolino, John V., and W. Michael McCracken. 1984. "Software test and evaluation in the Department of Defense." *Journal of Test and Evaluation*, V, 3 (October): 37-40.

Booch, Grady. 1983. *Software engineering with Ada.* Reading, MA: Benjamin/Cummings.

Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai. 1985. *Specification of software quality attributes.* Rome, NY: Rome Air Development Center. RADC-TR-85-37.

Browne, J. C. and Mary Shaw. 1981. "Toward a scientific basis for software evaluation." In *Software metrics*, Perlis, A., F. Sayward, and M. Shaw, editors: 19-41.

Bruno, Giorgio, and Giuseppe Marchetto. 1986. "Process-translatable petri nets for the rapid prototyping of process control systems." *IEEE Transactions on Software Engineering*, SE-12, 2 (February): 346-357.

Bruns, Glenn R., Ira Forman, Susan L. Gerhart, and Michael Graf. 1986. *Design technology assessment: The statecharts approach*. Austin, TX: Microelectronics and Computer Technology Corporation. MCC Technical Report Number STP-107-86.

Card, David N., Victor E. Church, and William W. Agresti. 1986. "An empirical study of software design practices." *IEEE Transactions on Software Engineering*, SE-12, 2 (February): 264-271.

Choi, B., R. A. DeMillo, W. Du, and R. Stansifer. 1988. *Observing reusable Ada software components—techniques for recording and using operational histories*. West Lafayette, IN: Software Engineering Research Center, Purdue University. SERC-TR-18-P.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. 1986. *Software engineering metrics and models*. Reading, MA: Benjamin/Cummings.

Cristian, F. 1980. "Exception handling and software fault-tolerance." In *Digest of Papers: 10th international symposium on fault-tolerant computing systems (FTCS-10)*, Kyoto, October 1-3, 1980: 97-103.

Currit, A., M. Dyer, and H. D. Mills. 1986. "Certifying the reliability of software." *IEEE Transactions on Software Engineering*, SE-12, 1 (January): 3-11.

Darringer, J. A., and J. C. King. 1978. "Applications of symbolic execution to program testing." *IEEE Computer*, 11, 4 (April): 51-60.

DeMillo, Richard A., Richard J. Lipton, and Alan J. Perlis. 1979. "Social processes and proofs of theorems and programs." *Communications of the ACM*, 22, 5 (May): 271-280.

DeMillo, Richard A., Richard J. Lipton, and Federick G. Sayward. 1978. "Hints on test data selection: Help for the practicing programmer." *IEEE Computer*, 11, 4 (April): 34-41.

DeMillo, Richard A. and Michael J. Merritt. 1983. "Protocols for data security." *IEEE Computer*, 16, 2 (February): 39-54.

DeMillo, Richard A., W. M. McCracken, R. J. Martin, and John F. Passafiume. 1987. *Software testing and evaluation*. Reading, MA: Benjamin/Cummings.

DoD Computer Security Center. 1983. *Department of Defense trusted computer system evaluation criteria*. Washington, DC: US DoD.

Duran, J. W. and S. Ntafos. 1981. "A report on random testing." In *Proceedings of the fifth international conference on software engineering*, San Diego, CA, March 9-12, 1981: 179-183.

Eastport Study Group. 1985. *Summer Study 1985. A report to the Director, Strategic Defense Initiative Organization*. Marina Del Rey, CA: Eastport Study Group.

Fairley, Richard E. 1985. *Software Engineering Concepts*. New York: McGraw-Hill.

Fujii, Marilyn S. 1978. "A comparison of software assurance methods." In *Proceedings of the software quality and assurance workshop*, San Diego, CA, November 15-17, 1978: 27-32. New York: ACM.

Goel, Amrit L. 1985. "Software reliability models: Assumptions, limitations, and applicability." *IEEE Transactions on Software Engineering*, SE-11, 12 (December): 1411-1423.

Goodenough, J. B. 1975. "Exception handling: Issues and a proposed notion." *Communications of the ACM*, 18, 12 (December): 683-696.

Green, C. 1976. "The design of the PSI Program Synthesis System." In *Proceedings of the second international conference on software engineering*, San Francisco, CA, October 13-15, 1976: 4-18.

Henderson, Peter. 1986. "Functional programming, formal specification, and rapid prototyping." *IEEE Transactions on Software Engineering*, SE-12, 2 (February): 241-250.

Hecht, Herbert and Myron Hecht. 1986. "Software reliability in the system context." *IEEE Transactions on Software Engineering*, SE-12, 1 (January): 51-58.

Hoare, C. A. R. 1978. "Communicating sequential processes." *Communications of the ACM*, 21, 8 (August): 666-677.

Jackson, Michael A. 1982. *System development*. Englewood Cliffs, NJ: Prentice-Hall.

Knight, John C., Nancy G. Leveson, and Louis D. St. Jean. 1985. "A large scale experiment in N-version programming." In *Digest of papers: 15th annual international symposium on fault-tolerant computing (FTCS-15)*, Ann Arbor, MI, June 19-21, 1985: 135-139.

Krauser, E. W. and A. P. Mathur. 1988. *Mutant unification for improved vectorization*. West Lafayette, IN: Software Engineering Research Center, Purdue University. SERC-TR-14-P.

Krishna, C. M., K. G. Shin, and R. W. Butler. 1984. "Synchronization and fault masking in redundant real-time systems." In *Digest of Papers: 14th annual international symposium on fault-tolerant computing (FTCS-14)*, Kissimmee, FL, June 20-22, 1984: 152-157.

Leveson, Nancy G. and Peter R. Harvey. 1983. "Analyzing software safety." *IEEE Transactions on Software Engineering*, SE-9, 5 (September): 569-579.

Leveson, Nancy G. and Timothy J. Shimeall. 1983. "Safety assertion for process control systems." In *Digest of Papers: 13th annual international symposium on fault-tolerant computing (FTCS-13)*, Milan, Italy, June 28-30, 1983: 236-240.

Linger, R. C., H. D. Mills, and B. I. Witt. 1979. *Structured programming: Theory and practice*. Reading, MA: Addison-Wesley.

Linn, Joseph L., Cy D. Ardoin, Cathy J. Linn, Stephen H. Edwards, Michael R. Kappel, and John Salasin. 1988. *Strategic Defense Initiative architecture dataflow modeling technique, version 1.5*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2035.

Liskov, B. H. and A. Snyder. 1979. "Exception handling in CLU." *IEEE Transactions on Software Engineering*, SE-5, 6 (November): 546-558.

Martin, R. J. 1987. *C2 software development and acquisition study, final report*. Prepared for Air Force Electronic Systems Division. Washington, D.C.: National Security Industrial Association.

Peterson, J. 1977. "Petri nets." *ACM Computing Surveys*, 9, 3 (September): 223-252.

Redwine, Samuel T., Jr., Louise Giovane Becker, Ann B. Marmor-Squires, R. J. Martin, Sarah H. Nash, and William E. Riddle. 1984. *DoD related software technology requirements: Practices, and prospects for the future*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-1788.

Rich, D. and H. E. Shrobe. 1978. "Initial report on a LISP programmer's apprentice." *IEEE Transactions on Software Engineering*, SE-4, 6 (November): 456-467.

Richardson, Debra J. and Lori A. Clarke. 1985. "Partition analysis: A method combining testing and verification." *IEEE Transactions on Software Engineering*, SE-11, 12 (December): 1477-1490.

Rine, David C. 1980. "Some models for security and protection analysis based on possibility theory and fuzzy sets." In *CYBERSOFT 80, International symposium on cybernetics and software*, Namur, Belgium, September 9, 1980.

Roby, Clyde G., editor. 1985. *Proceedings of the first IDA workshop on formal specification and verification of Ada*, March 18-20, 1985. Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-146.

Rowland, John H. 1988. "Artificial systems for software engineering studies." In *Proceedings of the second workshop on software testing, verification, and analysis*, Banff, Canada, July 19-21, 1988: 80-88.

Ruth, G. 1978. "Protosystem I: An automatic programming system prototype." In *AFIPS conference proceedings*, 47, Anaheim, CA, June 5-8, 1978: 675-681.

Scott, R. K., J. W. Gault, D. F. McAllister, and J. Wiggs. 1984. "Experimental validation of six fault-tolerant software reliability models." In *Digest of Papers: 14th annual international symposium on fault-tolerant computing (FTCS-14)*, Kissimmee, FL, June 20-22, 1984: 102-107.

Shriver, Bruce and Peter Wegner, editors. 1987. *Research directions in object-oriented programming*. Cambridge, MA: MIT Press.

Teichrow, D. and E. Hershey. 1977. "PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems." *IEEE Transactions on Software Engineering*, SE-3, 1 (January): 41-48.

Walsh, Patrick Joseph. 1985. *A measure of test case effectiveness*. Ph.D. diss., T. J. Watson School of Engineering, Applied Science, and Technology, State University of New York at Binghamton, NY.

Winograd, S. 1962. *Computation in the presence of noise*. Cambridge, MA: MIT Press.

Wirth, Niklaus. 1971. "Program development by stepwise refinement." *Communications of the ACM*, 14, 4 (April): 221-227.

Yourdon, E. and L. Constantine. 1979. *Structured design: Fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ: Prentice-Hall.

Zave, Pamela and William Schell. 1986. "Salient features of an executable specification language and its environment." *IEEE Transactions on Software Engineering*, SE-12, 2 (February): 312-325.

Distribution List for IDA Paper P-2143

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John P. Solomond Director Ada Joint Program Office Room 3E114 The Pentagon Washington, D.C. 20301-3081	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
IIT Research Institute 4600 Forbes Blvd., Suite 300 Lanham, MD 20706 Attn. Ann Eustice	1
Mr. Bill Easton P.O. Box 192 Bluemont, VA 22012	1
Mr. John Faust Rome Air Development Center RADC/COTC Griffis AFB, NY 13441	1
Mr. Pete Fonash Defense Communications Agency 1860 Wiehle Av. Reston, VA 22090	1
Mr. Don Greenlee OSD-DOTE Room 1C730 The Pentagon Washington, D.C. 20301	1
National Computer Security Center Attn V45 Ms. Sarah Hadley Bldg. 911 Airport Square 11 Linthicum, MD 21090	1

NAME AND ADDRESS	NUMBER OF COPIES
National Computer Security Center Attn V45 Mr. George Hoover Bldg. 911 Airport Square 11 Linthicum, MD 21090	1
National Computer Security Center Attn V45 Ms. Kathy Kucherary Bldg. 911 Airport Square 11 Linthicum, MD 21090 (301) 859-4374	1
Dr. R. J. Martin 4045 North 300 West West Lafayette, IN 47906	1
Dr. Richard DeMillo 4045 North 300 West West Lafayette, IN 47906	1
National Computer Security Center Attn V45 Mr. Ken Rowe Bldg. 911 Airport Square 11 Linthicum, MD 21090	1
National Computer Security Center Attn V45 Mr. David Vaurio Bldg. 911 Airport Square 11 Linthicum, MD 21090	1
Ms. Christine Youngblut 17021 Sioux Ln. Gaithersburg, MD 20878	1
CSED Review Panel	
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Thomas C. Brandt 10302 Bluet Terrace Upper Marlboro, MD 20772	1
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1
Dr. Ernest W. Kent Philips Laboratories 345 Scarborogh Road Briarcliff Manor, NY 10510	1
Dr. John M. Palms, President Georgia State University University Plaza Atlanta, GA 30303	1
Mr. Keith Uncapher University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Mr. Bill R. Brykczynski, CSED	4
Ms. Anne Douville, CSED	1
Dr. Dennis Fife, CSED	1
Dr. Karen Gordon, CSED	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Richard J. Ivanetich, CSED	1
Mr. Michael R. Kappel, CSED	1
Mr. Terry Mayfield, CSED	1
Dr. Reginald N. Meeson, CSED	5
Ms. Katydean Price, CSED	2
Dr. Richard L. Wexelblat, CSED	1
IDA Control & Distribution Vault	3